CAPÍTULO 3

Estructura general de un programa

INTRODUCCIÓN

En los capítulos anteriores se ha visto la forma de diseñar algoritmos para resolver problemas con computadora. En este capítulo se introduce al proceso de la programación que se manifiesta esencialmente en los programas.

El *concepto de programa* como un conjunto de instrucciones y sus tipos constituye la parte fundamental del capítulo. La *descripción de los elementos básicos* de programación, que se encontrarán en casi todos los programas: interruptores, contadores, totalizadores, etc., junto con las normas elementales para la escritura de algoritmos y programas, conforman el resto del capítulo.

En el capítulo se examinan los importantes conceptos de datos, constantes y variables, expresiones, operaciones de asignación y la manipulación de las entradas y salidas de información, así como la realización de las funciones internas como elemento clave en el manejo de datos. Por último se describen reglas de escritura y de estilo para la realización de algoritmos y su posterior conversión en programas.

3.1. CONCEPTO DE PROGRAMA

Un *programa de computadora* es un conjunto de instrucciones —órdenes dadas a la máquina— que producirán la ejecución de una determinada tarea. En esencia, *un programa es un medio para conseguir un fin*. El fin será probablemente definido como la información necesaria para solucionar un problema.

El *proceso de programación* es, por consiguiente, un proceso de solución de problemas —como ya se vio en el Capítulo 2— y el desarrollo de un programa requiere las siguientes fases:

1. *definición y análisis del problema;*
2. *diseño de algoritmos:*
   - diagrama de flujo,
   - diagrama N-S,
   - pseudocódigo;
3. *codificación del programa;*
4. *depuración y verificación del programa;*
5. *documentación;*
6. *mantenimiento*.

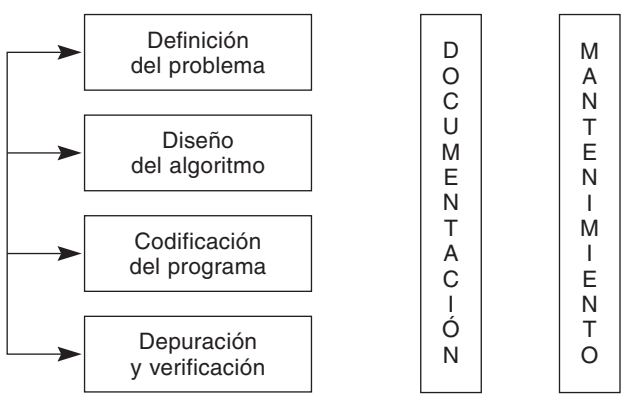


Figura 3.1. El proceso de la programación.

Las fases 1 y 2 ya han sido analizadas en los capítulos anteriores y son el objetivo fundamental de este libro; sin embargo, dedicaremos atención, a lo largo del libro (véase Capítulo 13) y en los apéndices, a las fases 3, 4, 5 y 6, aunque éstas son propias de libros específicos sobre lenguajes de programación.

3.2. PARTES CONSTITUTIVAS DE UN PROGRAMA

Tras la decisión de desarrollar un programa, el programador debe establecer el conjunto de especificaciones que debe contener el programa: *entrada*, *salida* y *algoritmos de resolución*, que incluirán las técnicas para obtener las salidas a partir de las entradas.

Conceptualmente un programa puede ser considerado como una caja negra, como se muestra en la Figura 3.2. La caja negra o el algoritmo de resolución, en realidad, es el conjunto de códigos que transforman las entradas del programa (*datos*) en salidas (*resultados*).

El programador debe establecer de dónde provienen las entradas al programa. Las entradas, en cualquier caso, procederán de un dispositivo de entrada —teclado, disco...—. El proceso de introducir la información de entrada —datos— en la memoria de la computadora se denomina *entrada de datos*, operación de *lectura* o acción de `leer`.

Las salidas de datos se deben presentar en dispositivos periféricos de salida: *pantalla*, *impresoras*, *discos*, etc. La operación de *salida de datos* se conoce también como *escritura* o acción de `escribir`.

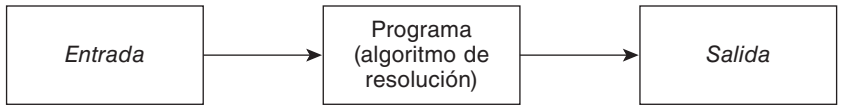


Figura 3.2. Bloques de un programa.

3.3. INSTRUCCIONES Y TIPOS DE INSTRUCCIONES

El proceso de diseño del algoritmo o posteriormente de codificación del programa consiste en definir las acciones o instrucciones que resolverán el problema.

Las *acciones* o *instrucciones* se deben escribir y posteriormente almacenar en memoria en el mismo orden en que han de ejecutarse, es decir, *en secuencia*.

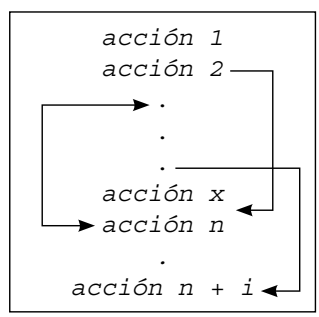
Un programa puede ser lineal o no lineal. Un programa es *lineal* si las instrucciones se ejecutan secuencialmente, sin bifurcaciones, decisión ni comparaciones.

```
instrucción 1
instrucción 2
      .
      .
      .
instrucción n
```

En el caso del algoritmo las instrucciones se suelen conocer como *acciones*, y se tendría:

```
acción 1
acción 2
    .
    .
    .
acción n
```

Un programa es *no lineal* cuando se interrumpe la secuencia mediante instrucciones de bifurcación.



3.3.1. Tipos de instrucciones

Las instrucciones disponibles en un lenguaje de programación dependen del tipo de lenguaje. Por ello, en este apartado estudiaremos las instrucciones —acciones— básicas que se pueden implementar de modo general en un algoritmo y que esencialmente soportan todos los lenguajes. Dicho de otro modo, las instrucciones básicas son independientes del lenguaje. La clasificación más usual, desde el punto de vista anterior, es:

1. *instrucciones de inicio/fin,*
2. *instrucciones de asignación,*
3. *instrucciones de lectura,*
4. *instrucciones de escritura,*
5. *instrucciones de bifurcación.*

Algunas de estas instrucciones se recogen en la Tabla 3.1.

Tabla 3.1. Instrucciones/acciones básicas

Tipo de instrucción	Pseudocódigo inglés	Pseudocódigo español
comienzo de proceso	`begin`	`inicio`
fin de proceso	`end`	`fin`
entrada (lectura)	`read`	`leer`
salida (escritura)	`write`	`escribir`
asignación	`A ← 5`	`B ← 7`

3.3.2. Instrucciones de asignación

Como ya son conocidas del lector, repasaremos su funcionamiento con ejemplos:

a) `A ← 80` la variable `A` toma el valor de `80`.
b) ¿Cuál será el valor que tomará la variable `C` tras la ejecución de las siguientes instrucciones?

```
A ← 12
B ← A
C ← B
```

`A` contiene `12`, `B` contiene `12` y `C` contiene `12`.

> ***Nota***
>
> Antes de la ejecución de las tres instrucciones, el valor de `A`, `B` y `C` es indeterminado. Si se desea darles un valor inicial, habrá que hacerlo explícitamente, incluso cuando este valor sea `0`. Es decir, habrá que definir e inicializar las instrucciones.

```
A ← 0
B ← 0
C ← 0
```

c) ¿Cuál es el valor de la variable `AUX` al ejecutarse la instrucción `5`?

```
1. A   ← 10
2. B   ← 20
3. AUX ← A
4. A   ← B
5. B   ← AUX
```

- en la instrucción `1, A` toma el valor `10`
- en la instrucción `2, B` toma el valor `20`
- en la instrucción `3, AUX` toma el valor anterior de `A`, o sea `10`
- en la instrucción `4, A` toma el valor anterior de `B`, o sea `20`
- en la instrucción `5, B` toma el valor anterior de `AUX`, o sea `10`
- tras la instrucción `5, AUX` sigue valiendo `10`.

d) ¿Cuál es el significado de `N ← N + 5` si `N` tiene el valor actual de `2`?

```
N ←   N + 5
```

Se realiza el cálculo de la expresión `N + 5` y su resultado `2 + 5 = 7` se asigna a la variable situada a la izquierda, es decir, `N` tomará un nuevo valor `7`.

Se debe pensar en la variable como en una posición de memoria, cuyo contenido puede variar mediante instrucciones de asignación (un símil suele ser un buzón de correos, donde el número de cartas depositadas en él variará según el movimiento diario del cartero de introducción de cartas o del dueño del buzón de extracción de dichas cartas).

3.3.3. Instrucciones de lectura de datos (entrada)

Esta instrucción lee datos de un dispositivo de entrada. ¿Cuál será el significado de las instrucciones siguientes?

a) **leer** (NÚMERO, HORAS, TASA)

Leer del terminal los valores `NÚMERO`, `HORAS` y `TASAS`, archivándolos en la memoria; si los tres números se teclean en respuesta a la instrucción son `12325`, `32`, `1200`, significaría que se han asignado a las variables esos valores y equivaldría a la ejecución de las instrucciones.

```
NÚMERO ← 12325
HORAS  ← 32
TASA   ← 1200
```

b) **leer** (A, B, C)

Si se leen del terminal `100`, `200`, `300`, se asignarían a las variables los siguientes valores:

```
A = 100
B = 200
C = 300
```

3.3.4. Instrucciones de escritura de resultados (salida)

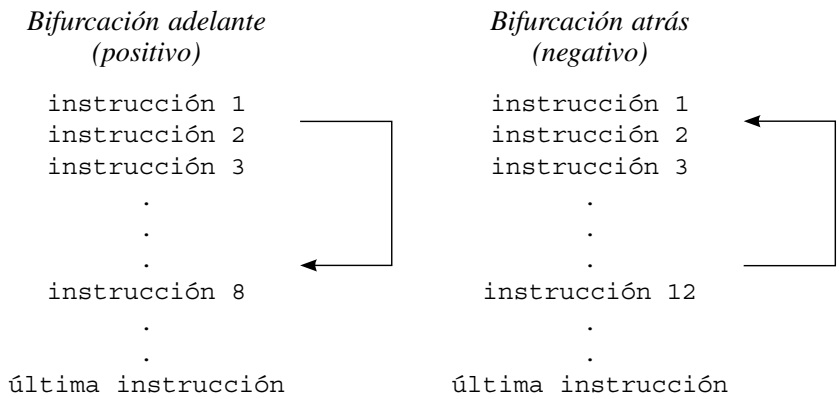
Estas instrucciones se escriben en un dispositivo de salida. Explicar el resultado de la ejecución de las siguientes instrucciones:

```
A ← 100
B ← 200
C ← 300
escribir (A, B, C)
```

Se visualizarían en la pantalla o imprimirían en la impresora los valores 100, 200 y 300 que contienen las variables `A`, `B` y `C`.

3.3.5. Instrucciones de bifurcación

El desarrollo lineal de un programa se interrumpe cuando se ejecuta una bifurcación. Las bifurcaciones pueden ser, según el punto del programa a donde se bifurca, hacia *adelante* o hacia *atrás*.



Las bifurcaciones en el flujo de un programa se realizarán de modo condicional en función del resultado de la evaluación de la condición.

Bifurcación incondicional: la bifurcación se realiza siempre que el flujo del programa pase por la instrucción sin necesidad del cumplimiento de ninguna condición (véase Figura 3.3).

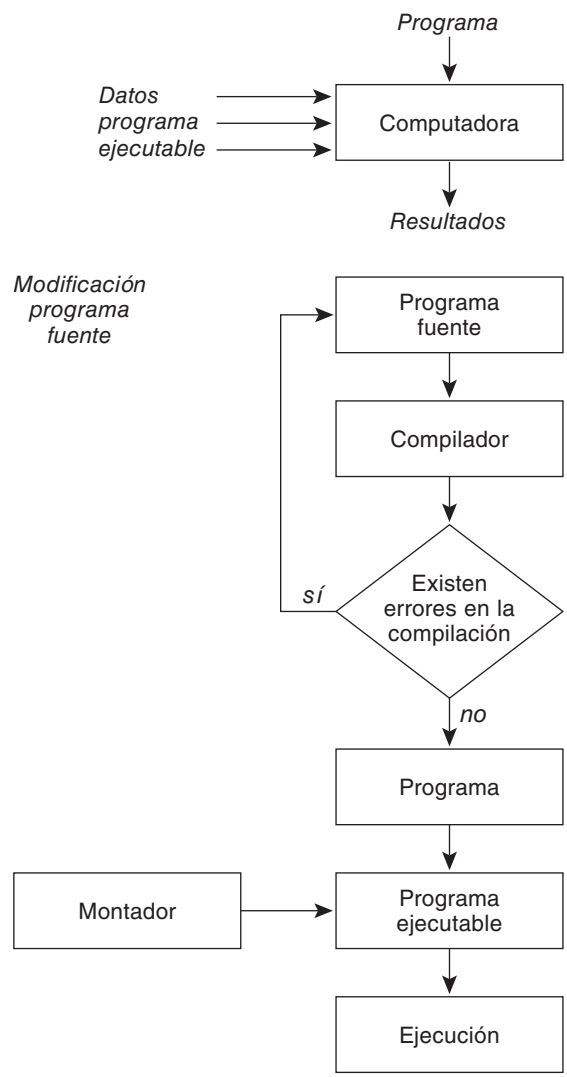


Figura 3.3. Fases de la ejecución de un programa.

Bifurcación condicional: la bifurcación depende del cumplimiento de una determinada condición. Si se cumple la condición, el flujo sigue ejecutando la acción F2. Si no se cumple, se ejecuta la acción F1 (véase Figura 3.4).

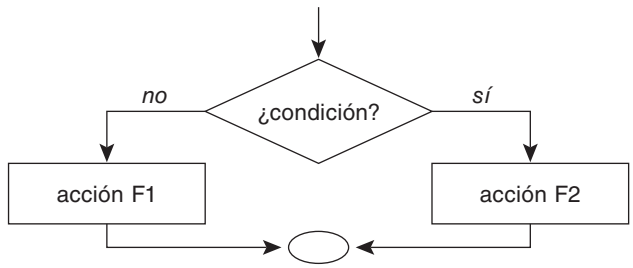


Figura 3.4. Bifurcación condicional.

3.4. ELEMENTOS BÁSICOS DE UN PROGRAMA

En programación se debe separar la diferencia entre el diseño del algoritmo y su implementación en un lenguaje específico. Por ello, se debe distinguir claramente entre los conceptos de programación y el medio en que ellos se implementan en un lenguaje específico. Sin embargo, una vez que se comprendan cómo utilizar los conceptos de programación y, la enseñanza de un nuevo lenguaje es relativamente fácil.

Los lenguajes de programación —como los restantes lenguajes— tienen elementos básicos que se utilizan como bloques constructivos, así como reglas para las que esos elementos se combinan. Estas reglas se denominan *sintaxis* del lenguaje. Solamente las instrucciones sintácticamente correctas pueden ser interpretadas por la computadora y los programas que contengan errores de sintaxis son rechazados por la máquina. Los elementos básicos constitutivos de un programa o algoritmo son:

- *palabras reservadas* (**`inicio`**, **`fin`**, **`si-entonces...`**, etc.),
- *identificadores* (nombres de variables esencialmente, procedimientos, funciones, nombre del programa, etc.),
- *caracteres especiales* (coma, apóstrofo, etc.),
- *constantes,*
- *variables,*
- *expresiones,*
- *instrucciones.*

Además de estos elementos básicos, existen otros elementos que forman parte de los programas, cuya comprensión y funcionamiento será vital para el correcto diseño de un algoritmo y naturalmente la codificación del programa. Estos elementos son:

- *bucles,*
- *contadores,*
- *acumuladores,*
- *interruptores,*
- *estructuras:*

  1. secuenciales,
  2. selectivas,
  3. repetitivas.

El amplio conocimiento de todos los elementos de programación y el modo de su integración en los programas constituyen las técnicas de programación que todo buen programador debe conocer.

3.5. DATOS, TIPOS DE DATOS Y OPERACIONES PRIMITIVAS

El primer objetivo de toda computadora es el manejo de la información o datos. Estos datos pueden ser las cifras de ventas de un supermercado o las calificaciones de una clase. Un *dato* es la expresión general que describe los objetos

con los cuales opera una computadora. La mayoría de las computadoras pueden trabajar con varios tipos (modos) de datos. Los algoritmos y los programas correspondientes operan sobre esos tipos de datos.

La acción de las instrucciones ejecutables de las computadoras se refleja en cambios en los valores de las partidas de datos. Los datos de entrada se transforman por el programa, después de las etapas intermedias, en datos de salida.

En el proceso de resolución de problemas el diseño de la estructura de datos es tan importante como el diseño del algoritmo y del programa que se basa en el mismo.

Un programa de computadora opera sobre datos (almacenados internamente en la memoria almacenados en medios externos como discos, memorias USB, memorias de teléfonos celulares, etc., o bien introducidos desde un dispositivo como un teclado, un escáner o un sensor eléctrico). En los lenguajes de programación los datos deben de ser de un *tipo de dato específico.* El tipo de datos determina cómo se representan los datos en la computadora y los diferentes procesos que dicha computadora realiza con ellos.

Tipo de datos

Conjunto específico de valores de los datos y un conjunto de operaciones que actúan sobre esos datos.

Existen dos tipos de datos: *básicos, incorporados o integrados (*estándar) que se incluyen en los lenguajes de programación; *definidos por el programador o por el usuario*.

Además de los datos básicos o simples, se pueden construir otros datos a partir de éstos, y se obtienen los datos compuestos o datos agregados, tales como **estructuras, uniones, enumeraciones** (*subrango*, como caso particular de las enumeraciones, al igual de lo que sucede en Pascal), **vectores** o **matrices/tablas** y **cadenas "arrays o arreglos"**; también existen otros datos especiales en lenguajes como C y C++, denominados **punteros (apuntadores)** y **referencias**.

Existen dos tipos de datos: *simples* (sin estructura) y *compuestos* (estructurados). Los datos estructurados se estudian a partir del Capítulo 6 y son conjuntos de partidas de datos simples con relaciones definidas entre ellos.

Los distintos tipos de datos se representan en diferentes formas en la computadora. A nivel de máquina, un dato es un conjunto o secuencia de bits (dígitos 0 o 1). Los lenguajes de alto nivel permiten basarse en abstracciones e ignorar los detalles de la representación interna. Aparece el concepto de tipo de datos, así como su representación. Los tipos de datos básicos son los siguientes:

numéricos *(entero, real)*
lógicos *(boolean)*
carácter *(caracter, cadena)*

Existen algunos lenguajes de programación —FORTRAN esencialmente— que admiten otros tipos de datos: **complejos**, que permiten tratar los números complejos, y otros lenguajes —Pascal— que también permiten declarar y definir sus propios tipos de datos: **enumerados** (*enumerated*) y **subrango** (*subrange*).

3.5.1. Datos numéricos

El tipo *numérico* es el conjunto de los valores numéricos. Estos pueden representarse en dos formas distintas:

- tipo numérico *entero (integer)*.
- tipo numérico *real (real)*.

Enteros: el tipo entero es un subconjunto finito de los números enteros. Los enteros son números completos, no tienen componentes fraccionarios o decimales y pueden ser negativos o positivos. Ejemplos de números enteros son:

```
   5          6
 -15          4
  20         17
1340         26
```

Los números enteros se pueden representar en 8, 16 o 32 bits, e incluso 64 bits, y eso da origen a una escala de enteros cuyos rangos dependen de cada máquina

Enteros	`-32.768`	*a*	`32.767`
Enteros cortos	`-128`	*a*	`127`
Enteros largos	`-2147483648`	*a*	`2147483647`

Además de los modificadores `corto` y `largo`, se pueden considerar `sin signo` (`unsigned`) y `con signo` (`signed`).

```
sin signo:    0 .. 65.5350
              0 .. 4294967296
```

Los enteros se denominan en ocasiones números de punto o coma fija. Los números enteros máximos y mínimos de una computadora[1] suelen ser `–32.768` a `+32.767`. Los números enteros fuera de este rango no se suelen representar como enteros, sino como reales, aunque existen excepciones en los lenguajes de programación modernos como C, C++ y Java.

Reales: el tipo real consiste en un subconjunto de los números reales. Los números reales siempre tienen un punto decimal y pueden ser positivos o negativos. Un número real consta de un entero y una parte decimal. Los siguientes ejemplos son números reales:

```
 0.08          3739.41
 3.7452         -52.321
-8.12             3.0
```

En aplicaciones científicas se requiere una representación especial para manejar números muy grandes, como la masa de la Tierra, o muy pequeños, como la masa de un electrón. Una computadora sólo puede representar un número fijo de dígitos. Este número puede variar de una máquina a otra, siendo ocho dígitos un número típico. Este límite provocará problemas para representar y almacenar números muy grandes o muy pequeños como son los ya citados o los siguientes:

```
4867213432   0.00000000387
```

Existe un tipo de representación denominado *notación exponencial* o *científica* y que se utiliza para números muy grandes o muy pequeños. Así,

```
367520100000000000000
```

se representa en notación científica descomponiéndolo en grupos de tres dígitos

```
367   520   100   000   000   000   000
```

y posteriormente en forma de potencias de 10

3.675201×10^{20}

y de modo similar

```
.0000000000302579
```

se representa como

3.02579×10^{-11}

[1] En computadoras de 16 bits como IBM PC o compatibles.

La representación en *coma flotante* es una generalización de notación científica. Obsérvese que las siguientes expresiones son equivalentes:

$3.675201 \times 10^{19} = .3675207 \times 10^{20} = .03675201 \times 10^{21} = \ldots$
$= 36.75201 \times 10^{18} = 367.5201 \times 10^{17} = \ldots$

En estas expresiones se considera la *mantisa* (parte decimal) al número real y el *exponente* (parte potencial) el de la potencia de diez.

`36.75201` *mantisa* `18` *exponente*

Los tipos de datos reales se representan en coma o punto flotante y suelen ser de simple precisión, doble precisión o cuádruple precisión y suelen requerir 4 bytes, 8 bytes o 10-12 bytes, respectivamente. La Tabla 3.2 muestra los datos reales típicos en compiladores C/C++.

Tabla 3.2. Tipos de datos reales (coma flotante) en el lenguaje C/C++

Tipo	Rango de valores
real (float)	**-3.4×10^{38} .. 3.4×10^{38}**
doble (double)	**-1.7×10^{-308} .. 1.7×10^{308}**

3.5.2. Datos lógicos (*booleanos*)

El tipo *lógico* —también denominado *booleano*— es aquel dato que sólo puede tomar uno de dos valores:

cierto o **verdadero** *(true)* y **falso** *(false)*.

Este tipo de datos se utiliza para representar las alternativas (*sí/no*) a determinadas condiciones. Por ejemplo, cuando se pide si un valor entero es par, la respuesta será verdadera o falsa, según sea par o impar.

C++ y Java soportan el tipo de dato *bool*.

3.5.3. Datos tipo carácter y tipo cadena

El tipo *carácter* es el conjunto finito y ordenado de caracteres que la computadora reconoce. Un dato tipo carácter contiene un solo carácter. Los caracteres que reconocen las diferentes computadoras no son estándar; sin embargo, la mayoría reconoce los siguientes caracteres alfabéticos y numéricos:

- caracteres alfabéticos `(A, B, C, ..., Z) (a, b, c, ..., z)`,
- caracteres numéricos `(1, 2, ..., 9, 0)`,
- caracteres especiales `(+, -, *, /, ^, ., ;, <, >, $, ...)`.

Una *cadena* (*string*) de *caracteres* es una sucesión de caracteres que se encuentran delimitados por una comilla (apóstrofo) o dobles comillas, según el tipo de lenguaje de programación. La *longitud* de una cadena de caracteres es el número de ellos comprendidos entre los separadores o limitadores. Algunos lenguajes tienen datos tipo *cadena*.

```
'Hola Mortimer'
'12 de octubre de 1492'
'Sr. McKoy'
```

3.6. CONSTANTES Y VARIABLES

Los programas de computadora contienen ciertos valores que no deben cambiar durante la ejecución del programa. Tales valores se llaman *constantes*. De igual forma, existen otros valores que cambiarán durante la ejecución del

programa; a estos valores se les llama *variables*. Una ***constante*** es un dato que permanece sin cambios durante todo el desarrollo del algoritmo o durante la ejecución del programa.

Constantes reales válidas

```
1.234
-0.1436

+ 54437324
```

Constantes reales no válidas

```
1,752.63  (comas no permitidas)
82  (normalmente contienen un punto decimal, aunque existen lenguajes que lo admiten sin punto)
```

Constantes reales en notación científica

`3.374562E` equivale a 3.374562×10^2

Una *constante tipo carácter* o *constante de caracteres* consiste en un carácter válido encerrado dentro de apóstrofos; por ejemplo,

```
'B'     '+'     '4'     ';'
```

Si se desea incluir el apóstrofo en la cadena, entonces debe aparecer como un par de apóstrofos, encerrados dentro de simples comillas.

```
""
```

Una secuencia de caracteres se denomina normalmente una *cadena* y una *constante tipo cadena* es una cadena encerrada entre apóstrofos. Por consiguiente,

```
'Juan Minguez'
```

y

```
'Pepe Luis Garcia'
```

son constantes de cadena válidas. Nuevamente, si un apóstrofo es uno de los caracteres en una constante de cadena, debe aparecer como un par de apóstrofos

```
'John"s'
```

Constantes lógicas (boolean)

Sólo existen dos constantes *lógicas* o *boolean*:

```
verdadero   falso
```

La mayoría de los lenguajes de programación permiten diferentes tipos de constantes: *enteras, reales, caracteres* y *boolean* o *lógicas*, y representan datos de esos tipos.

Una ***variable*** es un objeto o tipo de datos cuyo valor puede cambiar durante el desarrollo del algoritmo o ejecución del programa. Dependiendo del lenguaje, hay diferentes tipos de variables, tales como *enteras, reales, carácter, lógicas* y *de cadena*. Una variable que es de un cierto tipo puede tomar únicamente valores de ese tipo. Una variable de carácter, por ejemplo, puede tomar como valor sólo caracteres, mientras que una variable entera puede tomar sólo valores enteros.

Si se intenta asignar un valor de un tipo a una variable de otro tipo se producirá *un error de tipo*.

Una variable se identifica por los siguientes atributos: *nombre* que lo asigna y *tipo* que describe el uso de la variable.

Los nombres de las variables, a veces conocidos como ***identificadores***, suelen constar de varios caracteres alfanuméricos, de los cuales el primero normalmente es una letra. No se deben utilizar —aunque lo permita el lengua-

je, caso de FORTRAN— como nombres de identificadores palabras reservadas del lenguaje de programación. Nombres válidos de variables son:

```
A510
NOMBRES                  Letra            SalarioMes
NOTAS                    Horas            SegundoApellido
NOMBRE_APELLIDOS[2]      Salario          Ciudad
```

Los nombres de las variables elegidas para el algoritmo o el programa deben ser significativos y tener relación con el objeto que representan, como pueden ser los casos siguientes:

`NOMBRE`	para representar nombres de personas
`PRECIOS`	para representar los precios de diferentes artículos
`NOTAS`	para representar las notas de una clase

Existen lenguajes —Pascal— en los que es posible darles nombre a determinadas constantes típicas utilizadas en cálculos matemáticos, financieros, etc. Por ejemplo, las constantes $\pi = 3.141592...$ y $\mathbf{e} = 2.718228$ (base de los logaritmos naturales) se les pueden dar los nombres `PI` y `E`.

```
PI = 3.141592
E  = 2.718228
```

3.6.1. Declaración de constants y variables

Normalmente los identificadores de las variables y de las constantes con nombre deben ser declaradas en los programas antes de ser utilizadas. La sintaxis de la declaración de una variable suele ser:

```
<tipo_de_dato> <nombre_variable> [=<expresión>]
```

EJEMPLO

```
car letra, abreviatura
ent numAlumnos = 25
real salario = 23.000
```

Si se desea dar un nombre (identificador) y un valor a una constante de modo que su valor no se pueda modificar posteriormente, su sintaxis puede ser así:

```
const <tipo_de_dato> <nombre_constante> =<expresión>
```

EJEMPLO

```
const doble PI = 3.141592
const cad nombre = 'Mackoy'
const car letra = 'c'
```

3.7. EXPRESIONES

Las expresiones son combinaciones de constantes, variables, símbolos de operación, paréntesis y nombres de funciones especiales. Las mismas ideas son utilizadas en notación matemática tradicional; por ejemplo,

$$a + (b + 3) + \sqrt{c}$$

[2] Algunos lenguajes de programación admiten como válido el carácter subrayado en los identificadores.

Aquí los paréntesis indican el orden de cálculo y $\sqrt{\ }$ representa la función raíz cuadrada.

Cada expresión toma un valor que se determina tomando los valores de las variables y constantes implicadas y la ejecución de las operaciones indicadas. Una expresión consta de *operandos* y *operadores*. Según sea el tipo de objetos que manipulan, las expresiones se clasifican en:

- *aritméticas,*
- *relacionales,*
- *lógicas,*
- *carácter.*

El resultado de la expresión aritmética es de tipo numérico; el resultado de la expresión relacional y de una expresión lógica es de tipo lógico; el resultado de una expresión carácter es de tipo carácter.

3.7.1. Expresiones aritméticas

Las *expresiones aritméticas* son análogas a las fórmulas matemáticas. Las variables y constantes son numéricas (real o entera) y las operaciones son las aritméticas.

```
+                       suma
-                       resta
*                       multiplicación
/                       división
↑, **, ^                exponenciación
div, /                  división entera
mod, %                  módulo (resto)
```

Los símbolos +, –, *, ^ (↑ o **) y las palabras clave **div** y **mod** se conocen como *operadores aritméticos*. En la expresión

```
5 + 3
```

los valores `5` y `3` se denominan *operandos*. El valor de la expresión `5 + 3` se conoce como *resultado* de la expresión.

Los operadores se utilizan de igual forma que en matemáticas. Por consiguiente, `A · B` se escribe en un algoritmo como `A * B` y `1/4 · C` como `C/4`. Al igual que en matemáticas el signo menos juega un doble papel, como resta en `A – B` y como negación en `–A`.

Todos los operadores aritméticos no existen en todos los lenguajes de programación; por ejemplo, en FORTRAN no existe **div** y **mod**. El operador exponenciación es diferente según sea el tipo de lenguaje de programación elegido (^, ↑ en BASIC, ** en FORTRAN).

Los cálculos que implican tipos de datos reales y enteros suelen dar normalmente resultados del mismo tipo si los operandos lo son también. Por ejemplo, el producto de operandos reales produce un real (véase Tabla 3.3).

EJEMPLO

`5 x 7`	se representa por	`5 * 7`
$\frac{6}{4}$	se representa por	`6/4`
3^7	se representa por	`3^7`

Tabla 3.3. Operadores aritméticos

Operador	Significado	Tipos de operandos	Tipo de resultado
+	Signo positivo	Entero o real	Entero o real
-	Signo negativo	Entero o real	Entero o real
*	Multiplicación	Entero o real	Entero o real
/	División	Real	Real
div, /	División entera	Entero	Entero
mod, %	Módulo (resto)	Entero	Entero
++	Incremento	Entero	Entero
--	Decremento	Entero	Entero

Operadores DIV (/) y MOD (%)

El símbolo / se utiliza para la división real y la división entera (el operador **div** —en algunos lenguajes, por ejemplo BASIC, se suele utilizar el símbolo \— representa la división entera). El operador **mod** representa el resto de la división entera, y la mayoría de lenguajes utilizan el símbolo %.

```
A div B
```

Sólo se puede utilizar si A y B son expresiones enteras y obtiene la parte entera de A/B. Por consiguiente,

```
19 div 6                19/6
```

toma el valor 3. Otro ejemplo puede ser la división 15/6

```
15    |6
 3     2  cociente
  |
resto
```

En forma de operadores resultará la operación anterior

```
15 div 6 = 2            15 mod 6 = 3
```

Otros ejemplos son:

```
19 div 3         equivale a 6
19 mod 6         equivale a 1
```

EJEMPLO 3.1

Los siguientes ejemplos muestran resultados de expresiones aritméticas:

expresión	resultado	expresión	resultado
10.5/3.0	3.5	10/3	3
1/4	0.25	18/2	9
2.0/4.0	0.5	30/30	1
6/1	6.0	6/8	0
30/30	1.0	10%3	1
6/8	0.75	10%2	0

Operadores de incremento y decremento

Los lenguajes de programación C/C++, Java y C# soportan los operadores unitarios (unarios) de incremento, **++**, y decremento, **--**. El operador de incremento (++) aumenta el valor de su operando en una unidad, y el operador de decremento (--) disminuye también en una unidad. El valor resultante dependerá de que el operador se emplee como prefijo o como sufijo (antes o después de la variable). Si actúa como prefijo, el operador cambia el valor de la variable y devuelve este nuevo valor; en caso contrario, si actúa como sufijo, el resultado de la expresión es el valor de la variable, y después se modifica esta variable.

++i	Incrementa `i` en `1` y después utiliza el valor de `i` en la correspondiente expresión.
i++	Utiliza el valor de `i` en la expresión en que se encuentra y después se incrementa en `1`.
--i	Decrementa `i` en `1` y después utiliza el nuevo valor de `i` en la correspondiente expresión.
i-i--	Utiliza el valor de `i` en la expresión en que se encuentra y después se incrementa en `1`.

EJEMPLO:

```
n = 5
escribir n
escribir n++
escribir n
n = 5
escribir n
escribir ++n
escribir n
```

Al ejecutarse el algoritmo se obtendría:

```
5
5
6
5
6
6
```

3.7.2. Reglas de prioridad

Las expresiones que tienen dos o más operandos requieren unas reglas matemáticas que permitan determinar el orden de las operaciones, se denominan reglas de *prioridad* o *precedencia* y son:

1. Las operaciones que están encerradas entre paréntesis se evalúan primero. Si existen diferentes paréntesis anidados (interiores unos a otros), las expresiones más internas se evalúan primero.
2. Las operaciones aritméticas dentro de una expresión suelen seguir el siguiente orden de prioridad:
   - operador ( )
   - operadores ++, – – + y – unitarios,
   - operadores *, /, % (producto, división, módulo)
   - operadores +, – (suma y resta).

En los lenguajes que soportan la operación de exponenciación, este operador tiene la mayor prioridad.

En caso de coincidir varios operadores de igual prioridad en una expresión o subexpresión encerrada entre paréntesis, el orden de prioridad en este caso es de izquierda a derecha, y a esta propiedad se denomina *asociatividad*.

EJEMPLO 3.2

¿Cuál es el resultado de las siguientes expresiones?

a) 3 + 6 * 14

b) 8 + 7 * 3 + 4 * 6

Solución

a)
```
3 + 6 * 14
  3   + 84
      87
```

b)
```
8 + 7 * 3 + 4 * 6
  8   + 21     24
      29     + 24
             53
```

EJEMPLO 3.3

Obtener los resultados de las expresiones:

–4 * 7 + 2 ^ 3 / 4 – 5

Solución

```
-4 * 7 + 2 ^ 3 / 4 - 5
```

resulta

```
-4 * 7 + 8 / 4 - 5
-28 + 8 / 4 - 5
-28 + 2 - 5
-26 - 5
-31
```

EJEMPLO 3.4

Convertir en expresiones aritméticas algorítmicas las siguientes expresiones algebraicas:

$5 \cdot (x + y)$

$a^2 + b^2$

$\dfrac{x + y}{u + \dfrac{w}{a}}$

$\dfrac{x}{y} \cdot (z + w)$

Los resultados serán:

```
 5 * (x + y)
 a ^2 + b ^2
(x + y) / (u + w/a)
 x / y * (z + w)
```

EJEMPLO 3.5

Los paréntesis tienen prioridad sobre el resto de las operaciones:

`A * (B + 3)` la constante 3 se suma primero al valor de `B`, después este resultado se multiplica por el valor de `A`.

`(A * B) + 3`	`A y B` se multiplican primero y a continuación se suma 3.
`A + (B + C) + D`	esta expresión equivale a `A + B + C + D`
`(A + B/C) + D`	equivale a `A + B/C + D`
`A * B/C * D`	equivale a `((A * B)/C) * D` y no a `(A * B)/(C * D)`.

EJEMPLO 3.6

Evaluar la expresión `12 + 3 * 7 + 5 * 4`.

En este ejemplo existen dos operadores de igual prioridad, * (multiplicación); por ello los pasos sucesivos son:

```
12 + 3 * 7 + 5 * 4
     21
12 + 21 + 5 * 4
          20
12 + 21 + 20 = 53
```

3.7.3. Expresiones lógicas (*booleanas*)

Un segundo tipo de expresiones es la *expresión lógica* o *booleana*, cuyo valor es siempre verdadero o falso. Recuerde que existen dos constantes lógicas, *verdadera* (*true*) y *falsa* (*false*) y que las variables lógicas pueden tomar sólo estos dos valores. En esencia, una *expresión lógica* es una expresión que sólo puede tomar estos dos valores, *verdadero* y *falso*. Se denominan también *expresiones booleanas* en honor del matemático británico George Boole, que desarrolló el Álgebra lógica de Boole.

Las expresiones lógicas se forman combinando constantes lógicas, variables lógicas y otras expresiones lógicas, utilizando los *operadores lógicos* **`not`**, **`and`** y **`or`** y los *operadores relacionales* (de relación o comparación) `=, <, >, <=, >=, <>`.

Operadores de relación

Los operadores relacionales o de relación permiten realizar comparaciones de valores de tipo numérico o carácter. Los operadores de relación sirven para expresar las condiciones en los algoritmos. Los operadores de relación se recogen en la Tabla 3.4. El formato general para las comparaciones es

expresión1 **operador de relación** *expresión2*

y el resultado de la operación será verdadero o falso. Así, por ejemplo, si `A = 4` y `B = 3`, entonces

`A > B` *es verdadero*

Tabla 3.4. Operadores de relación

Operador	Significado
`<`	menor que
`>`	mayor que
`=, ==`	igual que
`<=`	menor o igual que
`>=`	mayor o igual que
`<>, !=`	distinto de

mientras que

```
(A - 2) < (B - 4)          es falso.
```

Los operadores de relación se pueden aplicar a cualquiera de los cuatro tipos de datos estándar: *enteros, real, lógico, carácter*. La aplicación a valores numéricos es evidente. Los ejemplos siguientes son significativos:

N1	N2	Expresión lógica	Resultado
3	6	3 < 6	verdadero
0	1	0 > 1	falso
4	2	4 = 2	falso
8	5	8 <= 5	falso
9	9	9 >= 9	verdadero
5	5	5 <> 5	falso

Para realizar comparaciones de datos tipo carácter, se requiere una secuencia de ordenación de los caracteres similar al orden creciente o decreciente. Esta ordenación suele ser alfabética, tanto mayúsculas como minúsculas, y numérica, considerándolas de modo independiente. Pero si se consideran caracteres mixtos, se debe recurrir a un código normalizado como es el ASCII (véase Apéndice A). Aunque no todas las computadoras siguen el código normalizado en su juego completo de caracteres, sí son prácticamente estándar los códigos de los caracteres alfanuméricos más usuales. Estos códigos normalizados son:

- Los caracteres especiales #, %, $, (,), +, –, /, . . ., exigen la consulta del código de ordenación.
- Los valores de los caracteres que representan a los dígitos están en su orden natural. Esto es, '0'<'1', '1'<'2', . . ., '8'<'9'.
- Las letras mayúsculas A a Z siguen el orden alfabético ('A'<'B', 'C'<'F', etc.).
- Si existen letras minúsculas, éstas siguen el mismo criterio alfabético ('a'<'b', 'c'<'h', etc.).

En general, los cuatro grupos anteriores están situados en el código ASCII en orden creciente. Así, '1'<'A' y 'B'<'C'. Sin embargo, para tener completa seguridad será preciso consultar el código de caracteres de su computadora (normalmente, el **ASCII**, *American Standar Code for Information Interchange* o bien el ambiguo código **EBCDIC**, *Extended Binary-Coded Decimal Interchange Code*, utilizado en computadoras IBM diferentes a los modelos PC y PS/2).

Cuando se utilizan los operadores de relación, con valores lógicos, la constante *false (falsa)* es menor que la constante *true (verdadera)*.

```
false < true
true > false
```

Si se utilizan los operadores relacionales = y <> para comparar cantidades numéricas, es importante recordar que la mayoría de los *valores reales no pueden ser almacenados exactamente*. En consecuencia, las expresiones lógicas formales con comparación de cantidades reales con (=), a veces se evalúan como falsas, incluso aunque estas cantidades sean algebraicamente iguales. Así,

```
(1.0 / 3.0) * 3.0 = 1.0
```

teóricamente es verdadera y, sin embargo, al realizar el cálculo en una computadora se puede obtener .999999. . . y, en consecuencia, el resultado es falso; esto es debido a la precisión limitada de la aritmética real en las computadoras. Por consiguiente, a veces *deberá excluir las comparaciones con datos de tipo real*.

Operadores lógicos

Los *operadores lógicos* o *booleanos* básicos son **not** *(no)*, **and** *(y)* y **or** *(o)*. La Tabla 3.5 recoge el funcionamiento de dichos operadores.

Tabla 3.5. Operadores lógicos

Operador lógico	Expresión lógica	Significado
no *(not)*, *!*	**no** p (*not* p)	negación de *p*
y *(and)*, *&&*	p **y** q (p *and* q)	conjunción de *p* y *q*
o *(o)*, *||*	p **o** q (p *o* q)	disyunción de *p* y *q*

Las definiciones de las operaciones **no**, **y**, **o** se resumen en unas tablas conocidas como *tablas de verdad.*

a	**no** a
verdadero	falso
falso	verdadero

no (6>10) es verdadera ya que (6>10) es falsa.

a	b	a **y** b
verdadero	verdadero	verdadero
verdadero	falso	falso
falso	verdadero	falso
falso	falso	falso

a **y** b es verdadera sólo si a y b son verdaderas.

a	b	a **o** b
verdadero	verdadero	verdadero
verdadero	falso	verdadero
falso	verdadero	verdadero
falso	falso	falso

a o b es verdadera cuando a, b o ambas son verdaderas.

En las expresiones lógicas se pueden mezclar operadores de relación y lógicos. Así, por ejemplo,

```
(1 < 5) y (5 < 10)        es verdadera
(5 > 10) o ('A' < 'B')    es verdadera, ya que 'A' < 'B'
```

EJEMPLO 3.7

La Tabla 3.6 resume una serie de aplicaciones de expresiones lógicas.

Tabla 3.6. Aplicaciones de expresiones lógicas

Expresión lógica	Resultado	Observaciones
(1 > 0) y (3 = 3)	verdadero	
no PRUEBA	verdadero	· PRUEBA es un valor lógico falso.
(0 < 5) **o** (0 > 5)	verdadero	
(5 <= 7) **y** (2 > 4)	falso	
no (5 <> 5)	verdadero	
(numero = 1) **o** (7 >= 4)	verdadero	· numero es una variable entera de valor 5.

Prioridad de los operadores lógicos

Los operadores aritméticos seguían un orden específico de prioridad cuando existía más de un operador en las expresiones. De modo similar, los operadores lógicos y relaciones tienen un orden de prioridad.

Tabla 3.7. Prioridad de operadores (lenguaje Pascal)

Operador	Prioridad
no *(not)*	más alta (primera ejecutada).
/, *, **div, mod, y** *(and)*	↓
+, -, o *(or)*	↓
<, >, =, <=, >=, <>	más baja (última ejecutada).

Tabla 3.8. Prioridad de operadores (lenguajes C, C++, C# y Java)

Operador	Prioridad
++ y -- (incremento y decremento en 1), +, –, !	más alta
*, /, % (módulo de la división entera)	↓
+, - (suma, resta)	↓
<, <=, >, >=	↓
== (igual a), != (no igual a)	↓
&& (**y** lógica, AND)	↓
|| (**o** lógica, or)	↓
=, +=, -=, *=, /=, %= (operadores de asignación)	más baja

Al igual que en las expresiones aritméticas, los paréntesis se pueden utilizar y tendrán prioridad sobre cualquier operación.

EJEMPLO 3.8

```
no 4 > 6                    produce un error, ya que el operador no se aplica a 4
no (4 > 14)                 produce un valor verdadero
(1.0 < x) y (x < z + 7.0)   si x vale 7 y z vale 4, se obtiene un valor verdadero
```

3.8. FUNCIONES INTERNAS

Las operaciones que se requieren en los programas exigen en numerosas ocasiones, además de las operaciones de las operaciones aritméticas básicas, ya tratadas, un número determinado de operadores especiales que se denominan *funciones internas*, incorporadas o estándar. Por ejemplo, la función **ln** se puede utilizar para determinar el logaritmo neperiano de un número y la función **raiz2 (sqrt)** calcula la raíz cuadrada de un número positivo. Existen otras funciones que se utilizan para determinar las funciones trigonométricas.

La Tabla 3.9 recoge las funciones internas más usuales, siendo x el argumento de la función.

Tabla 3.9. Funciones internas

Función	Descripción	Tipo de argumento	Resultado
abs(x)	valor absoluto de x	entero o real	igual que argumento
arctan(x)	arco tangente de x	entero o real	real
cos(x)	coseno de x	entero o real	real

Tabla 3.9. Funciones internas *(continuación)*

Función	Descripción	Tipo de argumento	Resultado
exp(x)	exponencial de x	entero o real	real
ln(x)	logaritmo neperiano de x	entero o real	real
log10(x)	logaritmo decimal de x	entero o real	real
redondeo(x) *(round(x))**	redondeo de x	real	entero
seno(x) *(sin(x))**	seno de x	entero o real	real
cuadrado(x) *(sqr(x))**	cuadrado de x	entero o real	igual que argumento
raiz2(x) *(sqrt(x))**	raíz cuadrada de x	entero o real	real
trunc(x)	truncamiento de x	real	entero

* Terminología en inglés.

EJEMPLO 3.9

Las funciones aceptan argumentos reales o enteros y sus resultados dependen de la tarea que realice la función:

Expresión	Resultado
raiz2 (25)	5
redondeo (6.5)	7
redondeo (3.1)	3
redondeo (-3.2)	-3
trunc (5.6)	5
trunc (3.1)	3
trunc (-3.8)	-3
cuadrado (4)	16
abs (9)	9
abs (-12)	12

EJEMPLO 3.10

Utilizar las funciones internas para obtener la solución de la ecuación cuadrática $ax^2 + bx + c = 0$. Las raíces de la ecuación son:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

o lo que es igual:

$$x1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad x2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Las expresiones se escriben como

```
x1 = (-b + raiz2 (cuadrado(b) - 4 * a * c)) / (2 * a)
x2 = (-b - raiz2 (cuadrado(b) - 4 * a * c)) / (2 * a)
```

Si el valor de la expresión

```
raiz2 (cuadrado(b) - 4 * a * c)
```

es negativo se producirá un error, ya que la raíz cuadrada de un número negativo no está definida.

3.9. LA OPERACIÓN DE ASIGNACIÓN

La operación de asignación es el modo de almacenar valores a una variable. La operación de asignación se representa con el símbolo u operador ← (en la mayoría de los lenguajes de programación, como C, C++, Java, el signo de la operación asignación es =). La operación de asignación se conoce como *instrucción* o *sentencia* de asignación cuando se refiere a un lenguaje de programación. El formato general de una operación de asignación es

<nombre de la variable> ← *<expresión>*

expresión es igual a expresión, variable o constante

La flecha (operador de asignación) se sustituye en otros lenguajes por = (Visual Basic, FORTRAN), := (Pascal) o = (Java, C++, C#). Sin embargo, es preferible el uso de la flecha en la redacción del algoritmo para evitar ambigüedades, dejando el uso del símbolo = exclusivamente para el operador de igualdad.

La operación de asignación:

```
A ← 5
```

significa que a la variable A se le ha asignado el valor 5.

La acción de asignar es *destructiva*, ya que el valor que tuviera la variable antes de la asignación se pierde y se reemplaza por el nuevo valor. Así, en la secuencia de operaciones

```
A ← 25
A ← 134
A ← 5
```

cuando éstas se ejecutan, el valor último que toma A será 5 (los valores 25 y 134 han desaparecido).

La computadora ejecuta la sentencia de asignación en dos pasos. En el primero de ellos, el valor de la expresión al lado derecho del operador se calcula, obteniéndose un valor de un tipo específico. En el segundo caso, este valor se almacena en la variable cuyo nombre aparece a la izquierda del operador de asignación, sustituyendo al valor que tenía anteriormente.

```
X ← Y + 2
```

el valor de la expresión Y + 2 se asigna a la variable X.

Es posible utilizar el mismo nombre de variable en ambos lados del operador de asignación. Por ello, acciones como

```
N ← N + 1
```

tienen sentido; se determina el valor actual de la variable N, se incrementa en 1 y a continuación el resultado se asigna a la misma variable N. Sin embargo, desde el punto de vista matemático no tiene sentido N ← N + 1.

Las acciones de asignación se clasifican según sea el tipo de expresiones en: *aritméticas, lógicas* y *de caracteres.*

3.9.1. Asignación aritmética

Las expresiones en las operaciones de asignación son aritméticas:

```
AMN ← 3 + 14 + 8
```

se evalúa la expresión 3 + 14 + 8 y se asigna a la variable AMN, es decir, 25 será el valor que toma AMN

```
TER1 ← 14.5 + 8
TER2 ← 0.75 * 3.4
COCIENTE ← TER1/TER2
```

Se evalúan las expresiones `14.5 + 8` y `0.75 * 3.4` y en la tercera acción se dividen los resultados de cada expresión y se asigna a la variable `COCIENTE`, es decir, las tres operaciones equivalen a `COCIENTE ← (14.5 + 8)/(0.75 * 3.4)`.

Otro ejemplo donde se pueden comprender las modificaciones de los valores almacenados en una variable es el siguiente:

```
A ← 0          la variable A toma el valor 0
N ← 0          la variable N toma el valor 0
A ← N + 1      la variable A toma el valor 0 + 1, es decir 1.
```

El ejemplo anterior se puede modificar para considerar la misma variable en ambos lados del operador de asignación:

```
N ← 2
N ← N + 1
```

En la primera acción `N` toma el valor `2` y en la segunda se evalúa la expresión `N + 1`, que tomará el valor `2 + 1 = 3` y se asignará nuevamente a `N`, que tomará el valor `3`.

3.9.2. Asignación lógica

La expresión que se evalúa en la operación de asignación es lógica. Supóngase que `M`, `N` y `P` son variables de tipo lógico.

```
M ← 8 < 5
N ← M o (7 <= 12)
P ← 7 > 6
```

Tras evaluar las operaciones anteriores, las variables `M`, `N` y `P` tomarán los valores *falso, verdadero, verdadero.*

3.9.3. Asignación de cadenas de caracteres

La expresión que se evalúa es de tipo cadena:

```
x ← '12 de octubre de 1942'
```

La acción de asignación anterior asigna la cadena de caracteres `'12 de octubre de 1942'` a la variable tipo cadena `x`.

3.9.4. Asignación múltiple

Todos los lenguajes modernos admiten asignaciones múltiples y con combinaciones de operadores, además de la asignación única con el operador ← . Así se puede usar el operador de asignación (←) precedido por cualquiera de los siguientes operadores aritméticos: `+, -, *, /, %`. La sintaxis es la siguiente:

```
<nombre_variable> ← <variable> <operador> <expresión>
```

y es equivalente a:

```
variable operador ← expresión
```

EJEMPLO

```
c ← c + 5              equivale a          c +← 5
a ← a * (b + c)        equivale a          a *← b + c
```

o si lo prefiere utilizando el signo de asignación (=) de C, C++, Java o C#.

Caso especial

Los lenguajes C, C++, Java y C# permiten realizar múltiples asignaciones en una sola sentencia

```
a = b = c = d = e = n +35;
```

Tabla 3.10. Operadores aritméticos de asignación múltiple

Operador de asignación	Ejemplo	Operación	Resultado
Entero a = 3, b= 5, c = 4, d = 6, e = 10			
*=	a += 8	a = a+8	a = 11
-=	b -= 5	b = b-5	b = 0
*=	c *= 4	c = c*4	c = 16
/=	d /= 3	d = d/3	d = 2
%=	e %= 9	e = e%9	e = 1

3.9.5. Conversión de tipo

En las asignaciones no se pueden asignar valores a una variable de un tipo incompatible al suyo. Se presentará un error si se trata de asignar valores de tipo carácter a una variable numérica o un valor numérico a una variable tipo carácter.

EJEMPLO 3.11

¿Cuáles son los valores de `A`, `B` *y* `C` *después de la ejecución de las siguientes operaciones?*

```
A ← 3
B ← 4
C ← A + 2 * B
C ← C + B
B ← C - A
A ← B * C
```

En las dos primeras acciones `A` y `B` toman los valores `3` y `4`.

```
C ← A + 2 * B          la expresión A + 2 * B tomará el valor 3 + 2 * 4 = 3 + 8 = 11
C ← 11
```

La siguiente acción

```
C ← C + B
```

producirá un valor de `11 + 4 = 15`

```
C ← 15
```

En la acción `B ← C - A` se obtiene para `B` el valor `15 - 3 = 12` y por último:

```
A ← B * C
```

`A` tomará el valor `B * C`, es decir, `12 * 15 = 180`; por consiguiente, el último valor que toma `A` será `180`.

EJEMPLO 3.12

¿Cuál es el valor de x después de las siguientes operaciones?

```
x ← 2
x ← cuadrado(x + x)
x ← raiz2(x + raiz2(x) + 5)
```

Los resultados de cada expresión son:

```
x ← 2                              x toma el valor 2
x ← cuadrado(2 + 2)                x toma el valor 4 al cuadrado; es decir 16
x ← raiz2(16 + raiz2(16) + 5)
```

en esta expresión se evalúa primero **`raiz2(16)`**, que produce 4 y, por último, **`raiz2(16+4+5)`** proporciona **`raiz2(25)`**, es decir, 5. Los resultados de las expresiones sucesivas anteriores son:

```
x ← 2
x ← 16
x ← 5
```

3.10. ENTRADA Y SALIDA DE INFORMACIÓN

Los cálculos que realizan las computadoras requieren para ser útiles la *entrada* de los datos necesarios para ejecutar las operaciones que posteriormente se convertirán en resultados, es decir, *salida*.

Las operaciones de entrada permiten leer determinados valores y asignarlos a determinadas variables. Esta entrada se conoce como operación de **lectura** (*read*). Los datos de entrada se introducen al procesador mediante dispositivos de entrada (teclado, tarjetas perforadas, unidades de disco, etc.). La salida puede aparecer en un dispositivo de salida (pantalla, impresora, etc.). La operación de salida se denomina **escritura** (*write*).

En la escritura de algoritmos las acciones de lectura y escritura se representan por los formatos siguientes:

```
leer (lista de variables de entrada)
escribir (lista de variables de salida)
```

Así, por ejemplo:

```
leer (A, B, C)
```

representa la lectura de tres valores de entrada que se asignan a las variables `A`, `B` y `C`.

```
escribir ('hola Vargas')
```

visualiza en la pantalla —o escribe en el dispositivo de salida— el mensaje `'hola Vargas'`.

Nota 1

Si se utilizaran las palabras reservadas en inglés, como suele ocurrir en los lenguajes de programación, se deberá sustituir

```
leer      escribir
```

por

```
read      write o bien print
```

Nota 2

Si no se especifica el tipo de dispositivo del cual se leen o escriben datos, los dispositivos de E/S por defecto son el teclado y la pantalla.

3.11. ESCRITURA DE ALGORITMOS/PROGRAMAS

La escritura de un algoritmo mediante una herramienta de programación debe ser lo más clara posible y estructurada, de modo que su lectura facilite considerablemente el entendimiento del algoritmo y su posterior codificación en un lenguaje de programación.

Los algoritmos deben ser escritos en lenguajes similares a los programas. En nuestro libro utilizaremos esencialmente el lenguaje algorítmico, basado en pseudocódigo, y la estructura del algoritmo requerirá la lógica de los programas escritos en el lenguaje de programación estructurado; por ejemplo, Pascal.

Un algoritmo constará de dos componentes: *una cabecera de programa* y *un bloque algoritmo*. La *cabecera de programa* es una acción simple que comienza con la palabra **algoritmo**. Esta palabra estará seguida por el nombre asignado al programa completo. El *bloque algoritmo* es el resto del programa y consta de dos componentes o secciones: *las acciones de declaración* y *las acciones ejecutables*.

Las *declaraciones* definen o declaran las variables y constantes que tengan nombres. Las *acciones ejecutables* son las acciones que posteriormente deberá realizar la computación cuando el algoritmo convertido en programa se ejecute.

```
algoritmo
cabecera del programa
sección de declaración
sección de acciones
```

3.11.1. Cabecera del programa o algoritmo

Todos los algoritmos y programas deben comenzar con una cabecera en la que se exprese el identificador o nombre correspondiente con la palabra reservada que señale el lenguaje. En los lenguajes de programación, la palabra reservada suele ser **program**. En Algorítmica se denomina **algoritmo**.

```
algoritmo DEMO1
```

3.11.2. Declaración de variables

En esta sección se declaran o describen todas las variables utilizadas en el algoritmo, listándose sus nombres y especificando sus tipos. Esta sección comienza con la palabra reservada **var** (abreviatura de *variable*) y tiene el formato

```
var
  tipo-1 : lista de variables-1
  tipo-2 : lista de variables-2
    .
    .
  tipo-n : lista de variables-n
```

donde cada *lista de variables* es una variable simple o una lista de variables separadas por comas y cada *tipo* es uno de los tipos de datos básicos (**entero**, **real**, **char** o **boolean**). Por ejemplo, la sección de declaración de variables

```
var
  entera : Numero_Empleado
  real   : Horas
  real   : Impuesto
  real   : Salario
```

o de modo equivalente

```
var
  entera : Numero_Empleado
  real   : Horas, Impuesto, Salario
```

declara que sólo las tres variables `Hora`, `Impuesto` y `Salario` son de tipo real.

Es una buena práctica de programación utilizar nombres de variables significativos que sugieran lo que ellas representan, ya que eso hará más fácil y legible el programa.

También es buena práctica incluir breves comentarios que indiquen cómo se utiliza la variable.

```
var
  entera : Numero_Empleado // número de empleado
  real   : Horas,          // horas trabajadas
           Impuesto,       // impuesto a pagar
           Salario         // cantidad ganada
```

3.11.3. Declaración de constantes numéricas

En esta sección se declaran todas las constantes que tengan nombre. Su formato es

```
const
  pi     = 3.141592
  tamaño = 43
  horas  = 6.50
```

Los valores de estas constantes ya no pueden variar en el transcurso del algoritmo.

3.11.4. Declaración de constantes y variables carácter

Las constantes de carácter simple y cadenas de caracteres pueden ser declaradas en la sección del programa **const**, al igual que las constantes numéricas.

```
const
  estrella = '*'
  frase    = '12 de octubre'
  mensaje  = 'Hola mi nene'
```

Las variables de caracteres se declaran de dos modos:

1. Almacenar un solo carácter.

   ```
   var carácter : nombre, inicial, nota, letra
   ```

   Se declaran `nombre`, `inicial`, `nota` y `letra`, que almacenarán sólo un carácter.

2. Almacenar múltiples caracteres (*cadenas*). El almacenamiento de caracteres múltiples dependerá del lenguaje de programación. Así, en los lenguajes

   VB 6.0/VB .NET (VB, Visual Basic)

   ```
   Dim var1 As String
   Var1 = "Pepe Luis García Rodriguez"
   ```

   Pascal formato tipo **array** o **arreglo** (véase Capítulo 8).

   Existen algunas versiones de Pascal, como es el caso de Turbo Pascal, que tienen implementados un tipo de datos denominados *string* (cadena) que permite declarar variables de caracteres o de cadena que almacenan palabras compuestas de diferentes caracteres.

   ```
   var nombre : string[20];    en Turbo Pascal
   var cadena : nombre[20];    en pseudocódigo
   ```

3.11.5. Comentarios

La documentación de un programa es el conjunto de información interna externa al programa, que facilitará su posterior mantenimiento y puesta a punto. La documentación puede ser *interna* y *externa*.

La *documentación externa* es aquella que se realiza externamente al programa y con fines de mantenimiento y actualización; es muy importante en las fases posteriores a la puesta en marcha inicial de un programa. La *documentación interna* es la que se acompaña en el código o programa fuente y se realiza a base de comentarios significativos. Estos comentarios se representan con diferentes notaciones, según el tipo de lenguaje de programación.

Visual Basic 6 / VB .NET

1. Los comentarios utilizan un apóstrofe simple y el compilador ignora todo lo que viene después de ese carácter

   ```
   'Este es un comentario de una sola línea
   Dim Mes As String 'comentario después de una línea de código
   ...............
   ```

2. También se admite por guardar compatibilidad con versiones antiguas de BASIC y Visual Basic la palabra reservada `Rem`

   ```
   Rem esto es un comentario
   ```

C/C++ y C#

Existen dos formatos de comentarios en los lenguajes C y C++:

1. Comentarios de una línea (comienzan con el carácter //)

   ```
   // Programa 5.0 realizado por el Señor Mackoy
   // en  Carchelejo (Jaén)en las Fiestas de Agosto
   // de Moros y Cristiano
   ```

2. Comentarios multilínea (comienzan con los caracteres /* y terminan con los caracteres */, todo lo encerrado entre ambos juegos de caracteres son comentarios)

```
/* El maestro Mackoy estudió el Bachiller en el mismo Instituto donde dio clase
Don Antonio Machado, el poeta */
```

Java

1. Comentarios de una línea

```
// comentarios sobre la Ley de Protección de Datos
```

2. Comentarios multilíneas

```
/* El pueblo de Mr. Mackoy está en Sierra Mágina, y produce uno
de los mejores aceites de oliva del mundo mundial             */
```

3. Documentación de clases

```
/**
        Documentación de la clase
*/
```

Pascal

Los comentarios se encierran entre los símbolos

```
(*              *)
```

o bien

```
{               }
(* autor J.R. Mackoy *)
{subrutina ordenacion}
```

Modula-2

Los comentarios se encierran entre los símbolos

```
(*              *)
```

> ***Nota***
>
> A lo largo del libro utilizaremos preferentemente para representar nuestros comentarios los símbolos `//` y `/*`. Sin embargo, algunos autores de algoritmos, a fin de independizar la simbología del lenguaje, suelen representar los comentarios con corchetes (`[ ]`).

3.11.6. Estilo de escritura de algoritmos/programas

El método que seguiremos normalmente a lo largo del libro para escribir algoritmos será el descrito al comienzo del Apartado 3.11.

```
algoritmo identificador        //cabecera
// seccion de declaraciones
```